AFRL-IF-RS-TR-2001-272
Final Technical Report
January 2002

CONTINUOUS VALIDATION OF EVOLVING SYSTEMS

AverStar, Incorporated

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. D889

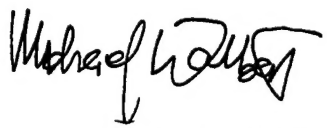*APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.*

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

20020507 098

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2001-272 has been reviewed and is approved for publication.

APPROVED: *Elizabeth S. Kean*

ELIZABETH S. KEAN
Project Engineer

FOR THE DIRECTOR: *Michael Talbert*

MICHAEL TALBERT, Maj., USAF, Technical Advisor
Information Technology Division
Information Directorate

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE JANUARY 2002	3. REPORT TYPE AND DATES COVERED Final Aug 96 - Jul 99

4. TITLE AND SUBTITLE
CONTINUOUS VALIDATION OF EVOLVING SYSTEMS

5. FUNDING NUMBERS
C - F30602-96-C-0217
PE - 62301E
PR - D889
TA - 01
WU - 01

6. AUTHOR(S)
Chris Garrity

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
AverStar, Incorporated
23 Fourth Avenue
Burlington Massachusetts 01803

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency   Air Force Research Laboratory/IFTD
3701 North Fairfax Drive   525 Brooks Road
Arlington Virginia 22203-1714   Rome New York 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

AFRL-IF-RS-TR-2001-272

11. SUPPLEMENTARY NOTES
Air Force Research Laboratory Project Engineer: Elizabeth S. Kean/IFTD/(315) 330-2601

12a. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
The objective of this effort was to develop a mission critical software capabilities package and integrate it with a consensus-based hyper-program framework. The languages to be supported in the prototype toolset included Ada95 and Java. The toolset was based on a framework consisting of standard World Wide Web technology and provided for easy import of existing Ada real-time applications so that large-scale software artifacts are available for early experimentation. The initial toolset included an Applet Import (Hypertexting) Tool and Demonstration.

14. SUBJECT TERMS
Software, Ada95, Java, Software Artifacts

15. NUMBER OF PAGES
48

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Standard Form 298 (Rev. 2-89) (EG)
Prescribed by ANSI Std. 239.18
Designed using Perform Pro, WHS/DIOR, Oct 94

Abstract

Under the EDCS Program, AverStar (formerly Intermetrics) investigated distributed debugging, testing and verification. The distributed application monitor technology was successfully transitioned to commercial tools. The primary result of this research is the flexible monitoring architecture shared by the monitoring tools. This architecture led to an architecture for continuous validation of dynamic component-based systems.

Table of Contents

Summary

Under the Evolutionary Design of Complex Software (EDCS) program, AverStar's focus was the creation of tools to facilitate dynamic programming on real DOD weapon systems. We created new capabilities for programming and debugging mixed-language software on heterogeneous distributed systems. Debugging, testing, and verification of these systems, during the course of their continuous evolution, turned out to be the largest challenge and the most fruitful area of research. Supporting these activities requires distributed test, verification, and debug support facilities, and we devised an architecture and a set of design methods for distributing the support facilities along with the application components.

A Bridge from Legacy Applications to Java

Most mission critical DOD systems are programmed in Ada, C, or C++, none of which are dynamic languages, and none of which have a strong orientation to distributed and internet-based applications. Java has the attributes we want, but has not yet made much headway among DOD application developers. To speed up the adoption of dynamic programming, continuous evolution, and internet-based distributed processing in DOD applications, we built a bridge between Ada and Java.

Our bridge between Ada and Java was a translator that converted Ada source code to Java bytecodes. This makes it possible to build a mixed Ada and Java system, and use the Java methods and concepts (such as Java Beans) for distributed communication. We determined that a C or C++ to Java bytecode translator, while potentially useful, would not be practical. C and C++ strive for programmer flexibility and freedom, and so allow many constructs that are unavailable in Java for safety and maintainability reasons. A C program can GO TO an array element, for example; in Java such actions are explicitly prohibited. Therefore, there would be no way to provide a full mapping from C (or C++) to Java bytecodes. Ada and Java are very compatible, however, and a full translation was provided.

This bridge (an Ada 95 to Java Bytecode translator) made it possible to apply the evolutionary, dynamically-bound, location-independent features of typical Java development to programs written in Ada.

Distributed Debugging, Testing, and Verification

We next turned our attention to the problems of debugging highly dynamic distributed systems. We created a system called Jade for instrumenting, monitoring, and analyzing distributed systems of Java (or Ada) applets. Jade was a distributed system debugging and analysis tool which assumed that the application consisted of components executing on heterogeneous computers connected by a variety of Internet communication services.

After the development of the Jade system it became clear that conventional debugging is too limited in the context of today's dynamic, distributed, component-based systems.

Traditional debugging takes place during development before an application is deployed, and the debugging information is usually removed before deploying for performance reasons. In a component-based system, elements of the system are upgraded dynamically after deployment. Therefore monitoring and analysis is needed before, during and after deployment of the application.

The technology that was developed under EDCS for Jade was the basis of the successful JWatch™ Java debugger and analysis tool. Furthermore, the Jade and JWatch technology have been extended under contract with NIST to produce EWatch™ for application event monitoring.

AverStar's EDCS research in these areas resulted in an architecture for monitoring highly dynamic systems that supports distributed component-based, Dynamic Object Oriented Language (DOOL), and legacy systems. It is also applicable to both real-time embedded and internet-wide applications. The remainder of this paper presents the rationale for building such systems; some lessons-learned, and our architecture recommendations. Further details about individual activities are in the appendices.

Distributed, Component-Based Applications

Consider the command and control software environments of today. The system may start as one well designed application that uses the graphical user interface technology of the time it is created. Over time, the system needs to evolve to include new hardware components or to handle new features, but because the existing system doesn't easily port to the new hardware and the existing application takes forever to modify, recertify and reinstall, a new application is written. The new application works on the new hardware, perhaps uses a different user interface style or technology, and makes some attempt to exchange data with the existing system. As more time passes more hardware and new requirements lead to a proliferation of applications and interfaces. One way to avoid this scenario is with similarly bad policy: never change to new hardware and never handle new or modified functionality.

Component-based approaches reduce the time for initial development, allow easy adoption of new hardware, and enable individual components to evolve over time to meet new requirements. However, these approaches also increase the complexity of debugging, testing, verification, and configuration management.

Distributed, component-based application problems are much more difficult to diagnose than client/server applications. Many more things can go wrong: security systems can prevent connections between components; contention for resources can lead to performance degradation and even frozen systems; the application can have a hidden dependencies on unknown components; and load-balancing middleware can create intermittent, hard-to-trace problems.

™ JWatch and EWatch are trademarks of AverStar, Inc.

2

Because software components are instantiated as needed and freed when their job is done, it is hard to find out exactly where pieces of an application are running. Multiple processes can share components, each holding a separate difficult-to-differentiate thread. Transactions can 'disappear' into the underlying component framework, returning only with unhelpful error messages. The root cause of problems can elude engineers for days. They lack the tools to pinpoint the application component that is causing or experiencing a problem. They need to get inside an application to understand it, learn which components are running and what those components are doing. Monitoring and diagnostic tools are essential for successful deployment of these systems.

Conventional debuggers rely on stopping the application to inspect data. This is problematic at best in a real-time distributed system. Conventional debuggers are weak in debugging multiple tasks even on a single computer. In real-time distributed systems, there are many computers running many tasks each, and the interactions among all of them have to be debugged.

Under the EDCS program, AverStar developed a suite of performance/timing monitoring and analysis tools. These tools, (the Jade system), support the debugging and performance tuning of distributed systems developed in Ada or Java, and using Internet protocols and commercial distributed processing APIs. Jade contains a very general mechanism for instrumenting distributed programs and the underlying system services they use (such as CORBA and DCOM). It produces detailed timing and synchronization profiles that are correlated back to the source code.

Jade works by inserting minimally intrusive probes into the executables on each machine. The probes collect data about the creation and destruction of objects, the synchronization of tasks, the use of I/O channels, and so on. Offline analysis tools process event logs and traces saved during execution, and help the user to visualize, explore, and understand the dynamics of application execution. Some aspects of the application can be abstracted into models for more sophisticated analysis by further tools.

Lessons-Learned

Evolution of large mission critical systems requires that the development environment, the hardware platform, the application software, and the supporting documentation all evolve in synch. In many cases, it is necessary for a heterogeneous collection of different versions to interoperate, sharing and updating common data and communications protocols even as the database schema and the message protocols evolve. It is rarely acceptable to shut an entire operation down whenever a new version of the software must be deployed.

AverStar, like much of the EDCS community, has the view that evolution of the development and maintenance environment will occur most smoothly if the programming environment based on accepted Web and Internet protocols and technologies

AverStar's participation in the EDCS program has given us the chance to explore a similar componentization of technology for testing, debugging, and monitoring. We have found that the dynamic Java environment on the web is an excellent match to our goals. By taking advantage of the powerful and growing dynamic Java infrastructure, we have been able to create capabilities that would have cost an order of magnitude more prior to the web revolution. We have also been helped enormously by our collaborations within the Dynamic Object Oriented Cluster of EDCS.

Products

Of the many potential paths to technology transfer, we have had the best results from incorporating the new technology in a package for OEM sales.

Rather than "marketing" our distributed, dynamic testing capabilities into other DOD and DARPA programs, or releasing a standalone product, we have arranged with the major commercial vendors of Java development tools to include our technology as an integral component of their product offerings.

The Jade system we developed under EDCS was further enhanced under a NIST cost-sharing (ATP) agreement. Then we invested our money in building a robust component that could be incorporated into other vendor's products.
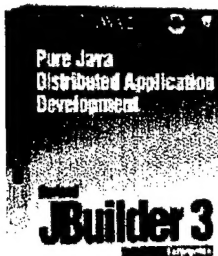
The descendents of Jade, **JWatch** and **EWatch,** are now available in retail stores worldwide. Borland and several other leading Java development tool vendors have incorporated our products in their commercial offerings. Everyone who purchases a Borland Java development environment gets JWatch and EWatch.. The marketing has all been done, and the technology is adopted by hundreds of new users every day.

Offering a product at retail, in computer stores, is the fastest way to get it widely adopted. However, the retail product business is very difficult to break into; retailers expect to be paid to stock the product, and paid again to give it favorable shelf space; retailers and publishers reap most of the profits, leaving very little to return to the developers. Therefore, capitalizing on the existing market presence and retailing "clout" of a company like Borland is extremely cost-effective, and yields wide market penetration quickly.

Additional Experiments

At the end of the EDCS project we explored the interaction between database schema evolution and configuration management in the context of distributed component-based applications on the World Wide Web. Incremental evolution of on-line distributed systems is a challenging and important problem. The following is a list of the problems we investigated:

Graphical user interfaces usually make assumptions about the structure of the information to be presented. These assumptions become invalid over time, but it is important to be able to view old information in a reasonable presentation format without having to "convert" very large databases.
To minimize network traffic, caching protocols must take advantage of opportunities to use slightly out of date versions as an alternative to always downloading the latest version of both code and data.
Users must be able to trust the behavior of the evolving system, which means that there must be strong protection against running inconsistent combinations of components.
For applications that use persistent objects, the object definitions, inheritance hierarchies and application code all must evolve without having to shut down running applications, and existing persistent data must continue to be usable after evolution.
Developers need the ability to test new components in the context of an operational system, without interfering with normal operations.
AverStar has faced problems similar to these many times before. For example, we do the configuration management for the Space Shuttle development environment, and we also built the configuration management system that GM-Delco uses for its engineering documentation world wide. Several aspects of the web environment, however, are unique. Because Java is a Dynamic Language, with individual components downloaded as necessary and dynamically linked, the potential for evolution and the potential for errors are both significantly greater. We have completed our first prototype addressing these issues: **Refresher™**.

The web environment is highly distributed. We have found that the web and Java infrastructures are powerful environments on which to build. We have been able to create quickly and at relatively low cost capabilities that would have been difficult or impossible to achieve in the past.

5

Developing a Diagnostic Architecture

The primary result of our EDCS work is the flexible monitoring architecture initially developed for Jade, and further extended with JWatch and EWatch. The Jade system was the basis of an extensible, event-driven framework for debugging Java applications. The
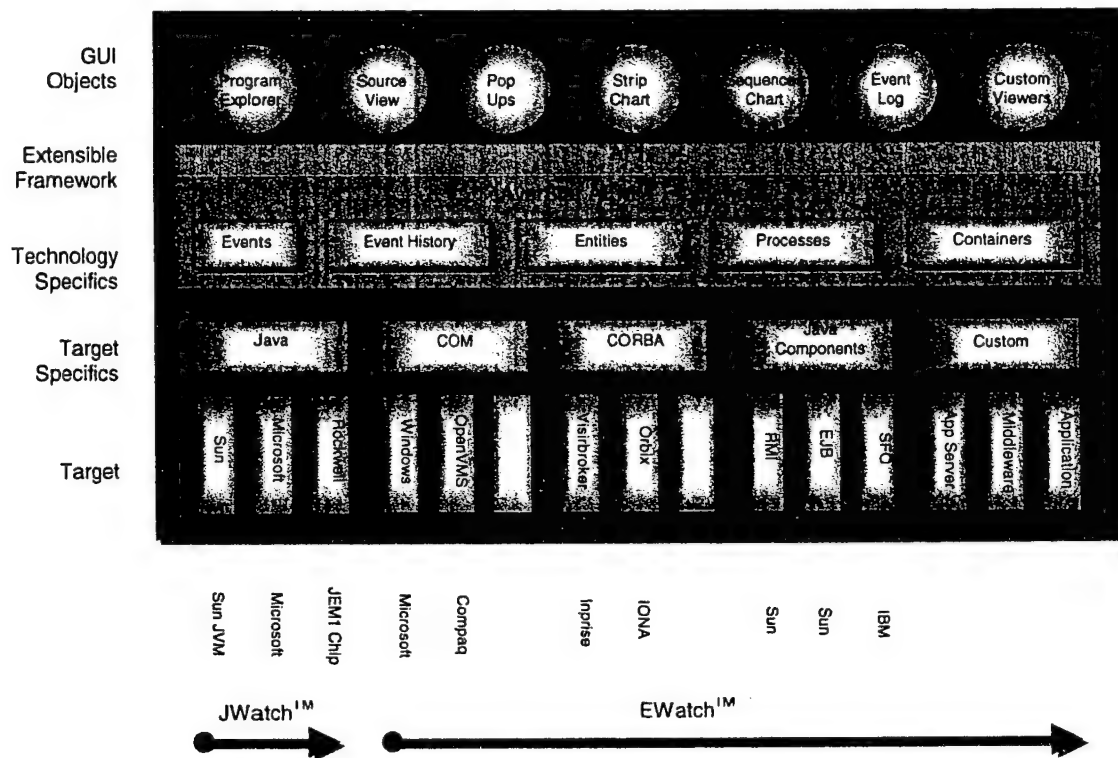
AverStar Application Event Monitoring Framework

key to the monitoring architecture is to insert the diagnostic probes into the middleware (COM, CORBA, etc.). The monitoring architecture then "underlays" the application's architecture, and monitoring occurs even when objects are transiently created and destroyed in the course of a transaction.

EWatch is a second-generation descendent of Jade. EWatch extends the JWatch system to support monitoring of CORBA, COM and Java components. EWatch monitors designated component events, and correlates and displays event information for use by developers, quality testers, systems integrators and application managers. Developers obtain debugging support, testers can analyze test execution and locate bottlenecks, system integrators can quickly tune and deploy their application, and application managers have a diagnostic tool that gets to the root cause of application problems.

6

EWatch complements enterprise management technology available today by providing end-to-end and point-to-point response measurement at the component level. It also provides insight into the business logic of application software.

As shown in the diagram below JWatch and EWatch share the same framework, and in fact share the same GUI.

A key to the success of both JWatch and EWatch is their architecture for monitoring applications. The monitoring architecture requires instrumentation of the *middleware* layer (shown above in the target layer) rather than the application itself. The Appendix *Toward an Architecture for Continuous Validation* discusses architectural implications of monitoring, debugging, and configuration management in a dynamic, distributed, real-time application.

Configuration management presents particular problems, because in these component-based systems the individual components are updated piecemeal. This is a strength of dynamic Java applications because the system does not have to be shut down while an upgrade is made. However, it presents a problem as well because often individual components are updated and integrated with an older version of another component that does not support the new interfaces.

A common scenario is that there is a user interface component that is communicating with an application server. Incremental modifications to the user interface to implement new features create multiple configurations of the application as a whole. Sometimes the user interface changes are compatible with the previous versions of the application server software, but sometimes new features in the system require changes in the server and interface to be updated in synch. Often the application is upgraded, or not upgraded in an ad hoc fashion, with the user never being aware of whether the upgrade was required or not.

AverStar created a prototype of a tool to manage Java applet configurations called Refresher. In the prototype, Refresher:

Kept track of the versions of compatible components in a configuration
Kept track of optional versus required updates
Kept its own cache of components to minimize applet start up times
Kept track of which components in a new configurations had changed (had new versions) from the versions in the cache in order to minimize download time when updating
In the case of optional configurations, Refresher gave the user a choice of updating or not updating
In the case of a required update, the user was informed that it was required

Refresher was used to maintain client configurations for the CAETI Wyndhaven™ project. It proved to be useful in keeping user's client applets in synch with changes to the server and to the client itself. Refresher itself was specific to Java applets, but the

strategy used by it could be applied to other domains. More information on Refresher can be found in the appendices.

A monitoring approach that does not apply to real-time applications is going to fall short in many DoD programs. Space and time constraints imposed by the real-time nature of the problem prevented insertion of extensive debugging information and communication in real-time application, and therefore limited the application of traditional debugging and monitoring technology. Even if debugging was supported during development, it was certainly removed before deployment. However, as more DoD applications, including real-time applications, become distributed and evolve over time, the diagnosis of problems becomes increasingly difficult without the appropriate tools to help.

As part of our EDCS work we investigated an approach to monitoring real-time distributed applications. Some of AverStar's development tools are in use on the F22 program, and with our familiarity with this project we used it as a model. The approach is discussed in more detail in the appendices. It builds upon the JWatch/EWatch approach of instrumenting the middleware layer rather than the applications itself, and it describes a unique approach for compressing the trace data in order to deal with the space constraints associated with this type of system. This approach has never been tried in practice, but we are confident that this type of monitoring would be helpful to modern real-time applications

Future directions

Reliable Evolution of Component-Based Systems

In the future, the DoD wants to meet most of its mission critical application needs using commercial off the shelf ("COTS") components, while adding additional components and error-detecting wrappers to meet fully its most demanding requirements. The vision is to integrate distributed application components using standard protocols and "middleware", while assuring that the damage from a failure in a component has only a local effect on that component's functionality. This vision offers many advantages including cost savings, shorter times to deploy systems, and flexibility, but significant technological challenges must be overcome to make it a reality.

Perhaps the biggest challenge is to assure that an assembly of components will continue to perform reliably as the components evolve, and as new functions are added. We anticipate that component specifications will include assertions defining expected preconditions and post-conditions, and that these semantic specifications will be analyzed statically and checked at run-time. Along with the application's logic, there will need to be continuous monitoring of system functionality and performance, and the data gathered will have to support fault detection and recovery.

Autonomous High Level Monitoring

The JWatch and EWatch systems are a huge step forward for analysis and debugging of distributed systems. However, they are not autonomous in the sense that they do not alert the user to anomalous behavior, but must be directed by a person to trace certain events, and then a person has to evaluate the output. Interface checking by the middleware is currently at the syntactic level, such as whether the right number of parameters were passed to a particular call. The current tracing is also at the level of individual calls and individual events.

The current framework could be used as the architecture of a high level monitoring facility. Instead of a person monitoring the events, the system would use a library of semantic-checking "fragments" to determine when to raise an alarm to a human monitor. The concept is simple, it is similar to conditional breakpoints in a traditional debugger, where the debugger continues past a breakpoint unless a certain condition (simple expression) is met. In the case of a dynamic distributed system, the system would log an alarm rather than creating a breakpoint. For example, if a query took longer than an acceptable time, or if a parameter value consistently exceeded a maximum value, an alarm would be raised. In these cases, the maintainer would want to be notified, in order to start probing the reason a response did not meet specifications, but you certainly wouldn't want to hit a breakpoint and stop a user from getting their response entirely. In the white paper included in the appendices we discuss the execution architecture for such a system; the types of high level semantic checking that could be applied this way; and

9

approaches to generate and manage the library of semantic-checking fragments for an application.

Data Evolutionary Systems

Most of the research focus of complex dynamic applications has been on the evolution of the software itself. As more component-based systems are developed and deployed, the tools are being developed that allow software engineers to create and debug these systems. Most knowledge-based systems, however, are data driven. The software is simply a "thin" interface on top of huge stores of data.

The evolution of data in data-driven applications has been largely ignored. In general the view is taken that updating the data is simply a database problem. When the schema of a database changes, access routines must be updated in synch, or be able to detect incompatibilities and recover gracefully. The addition or deletion of fields is relatively easy to spot, but changing the meaning of a field (e.g., changing the set of allowed values, or the meanings of those values) is more subtle and can easily go undetected. Changes to the schema can also invalidate old data, and require that the database go through a massive conversion process. This will usually result in downtime while the conversion happens. Our goal ought to be to support evolution of data in dynamic systems without requiring massive conversions or downtime. If individual software components can be replaced on the fly, then databases should similarly support changes to the schema on the fly.

Conclusions & Recommendations

AverStar has defined a three-tier debugging and monitoring architecture called the Application Event Monitoring (AEM) framework to support the reliable component at a time evolution of applications. AEM components complement the application-specific components. The AEM components provide a COTS solution to monitoring, provide a basis for error detection and fault recovery, and allow the addition of application specific capabilities. Tightly connected to the system being monitored are non-disruptive probes that efficiently recognize application events. One or more probes are supported and controlled by probe managers (the middle tier of the Application Event Monitor architecture). On the users' desks are consoles that allow personnel doing debugging, integration, and support to control the focus of attention, perform specific analyses, and record event streams and results. The console and probe manager software will be reusable as COTS components. Some probes, in particular probes that recognize messages between CORBA and/or DCOM component interfaces, will also be reusable, but special purpose probes will need to be constructed for DOD's real-time embedded components.

It is now time to test the hypothesis that applying this COTS debugging technology to a real-time embedded defense application is as straightforward as creating one or more special purpose probes. In addition, we must demonstrate that the event streams captured by the probes can be routed to other testing tools being created by the EDCS community, multiplying the power of the AEM architecture.

Appendices

Towards an Architecture for Continuous Validation

The creation of reliable applications out of components from anywhere is a major DoD goal. Achieving this goal requires new techniques for assuring the reliability of these applications as they are developed, and continuing to assure reliability as they are dynamically upgraded in the field.

The reliability of a system is much more than an aggregation of the reliability metrics of the individual components. In order to measure the reliability (or other quality metrics) for the system, we need to take cross-module, system-wide measurements, and determine application properties that cut across the components.

Without these global kinds of analysis, it is very difficult to get separately developed components to work together. It is easy for applications to pass the wrong information between components, or fail to account for undocumented assumptions made by components. Another common cause of problems is the failure to meet required timing constraints that are known, but not stated or verified. With these problems, DoD programs that require a high degree of assurance cannot rely on reused black box components, and cannot take full advantage of component sources outside the DoD such as the open source community. As a result, software development takes too long, costs too much, and produces unreliable systems.

The approach described in this paper supports monitoring and assessment of component suitability of a complex system during development and integration, during deployment in the field, and after deployment as components are changed and upgraded. It is not intended to provide support for choosing the appropriate components or composing them, but rather for monitoring and aiding in evaluating the results of composing interchangeable parts in order to assess their suitability.

In order to monitor the system, assessment "fragments" are created that check global aspects of the system. For example, was a response to this type of query sent within a specified amount of time, or if event X happened, was it followed by events Y and Z or event P. If any assessment fragment fails, an alarm can be raised. The system as a whole is not monitored by a single measure of overall correctness, but by a conglomeration of assessment fragments that match the particular versions of the components that comprise the system. When a component is replaced, the assessment fragments that pertain to it are also replaced. The composability of the monitoring architecture matches the composability of the system architecture.

The key components of this approach are:

Middleware-Based Probes and Traces

It is difficult, if not impossible, to predict up front which messages and events will be used in monitoring. Instrumenting the application code requires the developer to try and predict what information will be needed

13

and provide that information by modifying the application. If the middleware layer (e.g. CORBA, DCOM, runtime libraries) is instrumented instead, data can be gathered on every message and event that occurs and that is known to the middleware – essentially all inter-component interactions.

Event Synthesis

When very low level events are collected, it will be necessary at times to coalesce a set of low level events into a higher level event or transaction. We might take a series of events like "are you ready", "yes", "here's the data", "got it" and replace it with a single synthesized event like "confirmed transmission". Semantic monitors can work on any combination of low-level and synthesized events.

Distributed Semantic Monitors

These are the assessment fragments that check the validity of the system. They need to support a rich, open-ended set of semantic concepts, as discussed in the next section. These monitors comprise a language independent distributed representation for semantic consistency.

Flexible Repository of Monitors

Determining which semantic monitors to run on a particular configuration of application components will depend on matching patterns of events, matching message signatures, and matching the right version of the components involved. It will also be extremely dynamic as new monitors are added whenever a new version of a component is installed in the system. Therefore, a very flexible repository of semantic monitors is needed.

Data Evolution Models

Often a critical part of a distributed application is a knowledge base or database of some sort with an interface that is called by the rest of the application. The monitors will need to support the evolution of data in the system as well as software.

Application Architecture Model

Many of the high level semantic checks envisioned in this system rely on having knowledge of the overall application architecture. This knowledge will also need to be a component of the system in order to create the checks, and to know when it is appropriate to run them.

Execution Architecture

A distributed component-based system has pieces of the system spread across many physical processors. If a load-balancing application server is being used, the components may not be instantiated in predictable places on the network. Adding probes to the

component itself to trace events regardless of where it is instantiated is not possible if the component is an off-the-shelf product, or a legacy application. Based on the results of our earlier research, our solution to this problem is to instrument the middleware layer.

The monitoring architecture mimics the application's architecture. Each machine where a portion of the application may run has a probe installed on it. Host managers allow individual probes to be turned on or off. The probes generate the raw data for the validation analysis. The raw data could be analyzed in real-time on the application system, but in many cases the application's processor needs to be dedicated to servicing the application, in which case the probe data is sent to a separate machine for analysis.
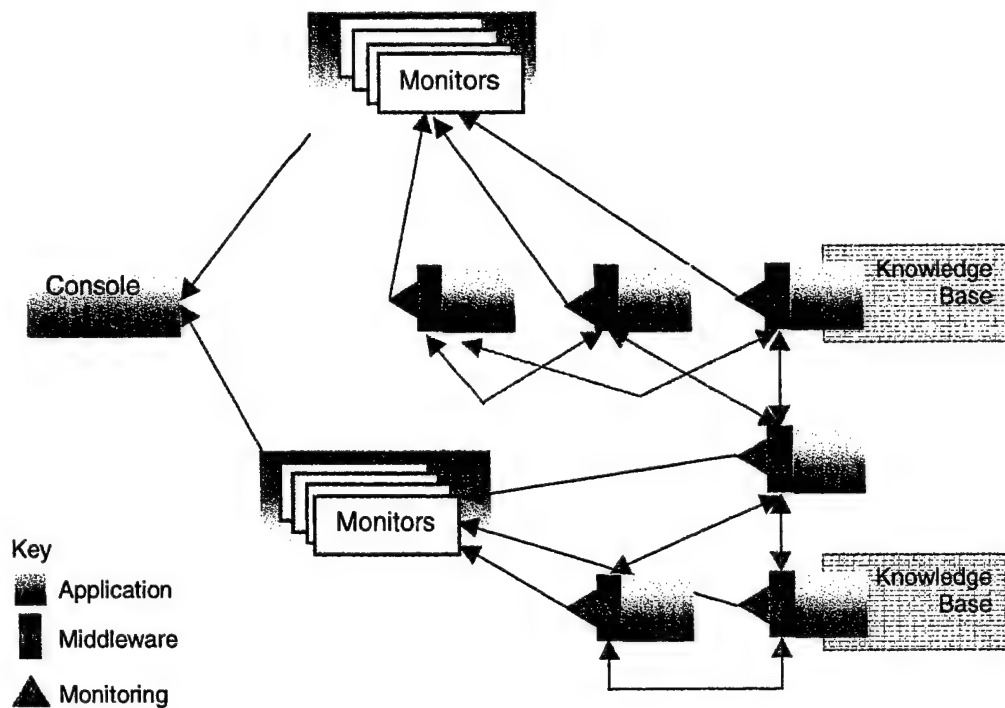

Figure 1 Monitoring Execution Architecture

In the above diagram objects in red represent the application. Communication between application processes is handled by the middleware shown in green. This could be a commercial middleware product like COM or CORBA, or it could be application specific in the case of real-time applications, like a real-time executive. The orange blocks represent knowledge bases accessed by the application. Monitoring is represented in blue.

Probes in each of the middleware layers capture event and message data. These program traces are low level and fine-grained. Program traces are sent to a monitoring processor. Based on pattern matching and knowledge of the system, the monitoring processor will

execute a set of distributed semantic monitors to assess the state of the system. Monitoring analysis takes place on processors that are generally separate from the applications processors, and any alerts are communicated to the console. Normal operation, i.e., within tolerances, would not clutter up the console. The user could however, request that additional events be reported to the console. When an alert is raised the console provides the user with the ability to drill down into the details of the system, and to look at the trace data in whatever level of detail they wish.

The assessment routines can be generated in a number of ways:

- Hand written, though this would be tedious and error prone for a large distributed system

- Captured or generated from design tools. For example, from UML constraints.

- Produced from system architecture models and architecture description languages. The high level knowledge and constraints expressed in these forms could be translated in to assessment fragments.

A flexible repository will be needed to store and manage these fragments. The fragments will have to match up with certain events or combinations of events. There will also need to be version management of fragments as a fragment that checks the parameters of one version of a data access routine may not perform the correct checks for the next version of the data access routine that includes additional data.

Semantic Monitoring

Current middleware and language systems will generally perform a level of syntax checking. If a method invocation expects three parameters and only two are passed this will generally be caught by current tools. The distributed semantic monitors introduced above need to be at a much higher level in order to provide real assurance of system parameters that cross components.

Distributed Semantic Monitors can contain any arbitrary logic – they are not limited to expressions. In some cases a simple expression ("is this parameter non-zero") will be fine, but in other cases (if event X occurs, then either events Y and Z must follow within three seconds, or event E must occur within one second) more logic will be needed. The types of checking provided include:

- Caller/callee version expectations are consistent. If when version 1.1 of a component A was implemented, it was consistent with versions 2.0 through 3.2 of component B, a checking fragment should be generated that checks whether the version of B is consistent with those that A expects. If B is a newer version, the call that A is making may still work, but an alert should occur to provide feedback that this is a potential error.

16

- Value constraint checks. At the syntactic level a value may simply be a number, but at a semantic level, the value may be a temperature, and the system may require the temperature to be above freezing. In one version this check may be that the value is greater than 32°, but a different version of the same component might need to check that the value is greater than 0° (centigrade) or greater than 273° (kelvin).

- Failure to follow a partially ordered set of events (POSETS). During design a POSET may be specified as part of the expected behavior of the system. Or during implementation or integration a pattern of events may be recognized and turned into a POSET to check against. Any failure to match the expected behavior of the system is probably an error that needs looking into.

- Response and other Quality of Service (QoS) constraints. Often it is not enough that event X follow event Y. To meet the performance goals of the system that response must occur within some limited span of time. These constraints are often specified and evaluated at design time. Particular hardware may be specified in order to meet performance requirements, but currently there is very little that will allow a user to pin point particular events as failing to meet performance constraints.

- Precision constraints. If the composition of two components requires a translation process between the components, for example, between centigrade and Fahrenheit, then the translation process produces a value that is within the acceptable bounds for rounding errors, or other inaccuracies in the translation.

- Data version consistency. If the caller is consistent with a certain version of a knowledge base, is the data also consistent with that version. If there is a newer version of the knowledge base is there a translation process, or does the caller get back data that it doesn't know how to deal with.

This architecture is not constraining to application developers. They can still build and buy components according to any desired application architecture. Our proposals concern an additional "overlay" on top of the application architecture, providing instrumentation in the middleware, probes and monitors, and Distributed Semantic Monitors. Widespread use of this architecture will enable us to incrementally build up a library of useful checking fragments (monitors); to carry monitors from one application to another; and to develop portable knowledge about the characteristics and risks of individual (black box) components.

JWatch™/EWatch™ Architecture

EWatch views the components of a distributed application as a set of interacting objects running on client and server machines, each firing and listening for events. Using Probes that intercept application events, EWatch correlates, interprets and displays the events in real-time so that programmers, QA personnel, system integrators and support staff can better understand the behavior of the system as it executes. Users can easily identify processes that hang, fail to execute properly or that cannot connect with a server. Timing
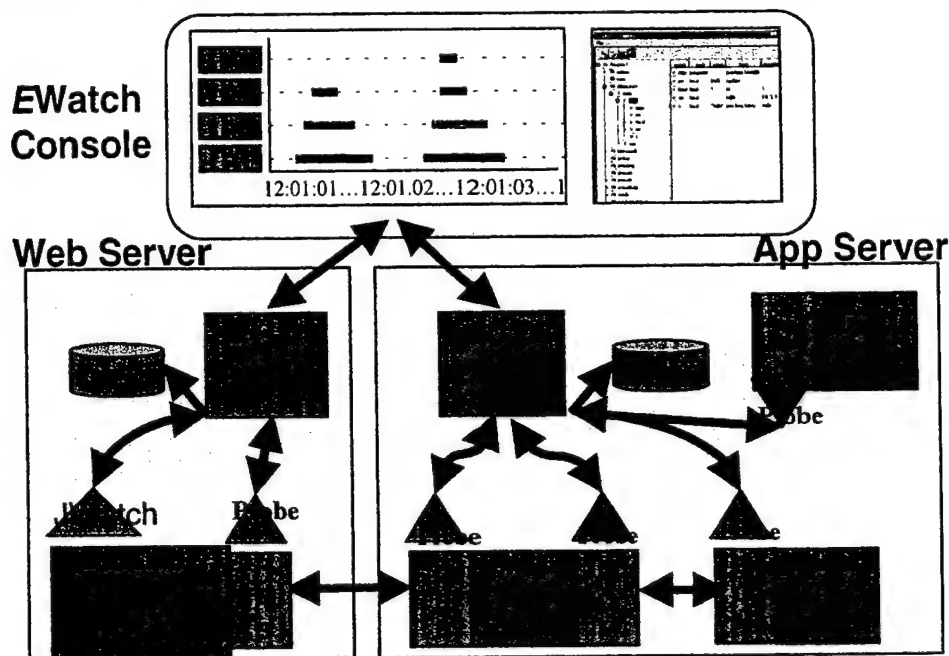

Figure 1. Distributed Application with EWatch Components In Place

information captured by the Probes helps pinpoint performance bottlenecks.

EWatch places its Probes at key points in CORBA, COM and Java technology-based applications. Probes can also be custom made to fit into other of new or existing technologies.

The electronic commerce application illustrated in Figure 1 exemplifies the environment that EWatch monitors. The Web Server is running a Java process (P1) inside a Java virtual machine. It communicates via CORBA's IIOP to an Application Server. The Application Server runs three processes (P2, P3 and P4) that communicate with one another using COM. Probes for CORBA and COM are activated in the application processes and watch for events on both the Web Server and the Application Server. The events are passed to the local Host Managers. The Host Managers store the events in a Log File and pass them to the Console for display in an animated Strip Chart (Figure 4)

18

and also a real-time Logview (Figure 5) that show the monitored processes and the application event data.

E *Watch Architecture*

Multi-tier Architecture

EWatch Probes, Host Manager, Server and Console form a multi-tier architecture (Figure 2) that provides optimal flexibility in deployment while minimizing overhead on the systems and network supporting the application.
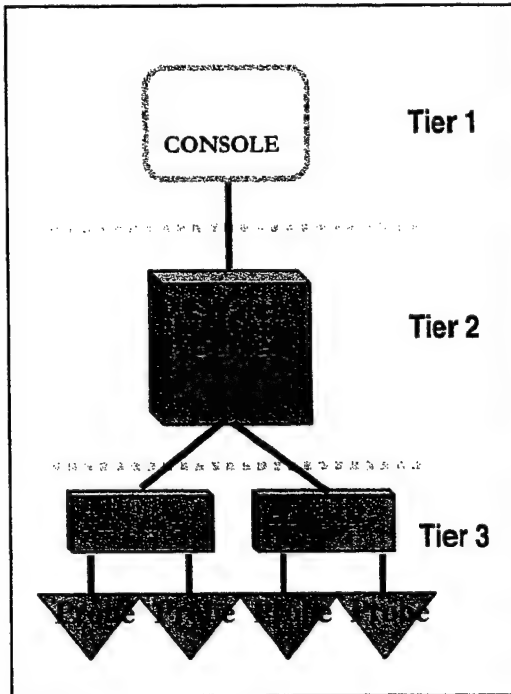

Figure 2 Multi-tier EWatch Architecture

EWatch Probes

EWatch Probes are specific to the technology and vendor implementation they target. They require no changes to source code or re-linking of the application binary code. A Probe Developer's Kit will allow partners to readily target a wide range of technologies. EWatch Probes and Host Managers can be installed as needed, or they can be distributed with an application as a monitoring and diagnostic subsystem. Probes and Host Managers remain latent until activated by an EWatch Console.

EWatch Host Manager

Each host being monitored has an EWatch Host Manager that collects information from the Probes and relays tracing configurations from the EWatch Server to its local Probes, directing them to look for specific types of events. The Host Managers are controlled from the Console via the EWatch Server. The Host Manager maintains a persistent Log File of events as the application executes. A Host Manager can manage any number of Probes for different component technologies.

EWatch Server

The EWatch Server can run on a system dedicated to system-monitoring functions or can be co-located with the Console or a Host Manager. The Server collects, correlates and interprets event information in an in-memory database. It transmits Probe configurations to Host Managers on target systems and event information to the Console for display.

19

EWatch Console

From the Console, users can configure the events to be captured by the Probes and can also define all or a subset of those events for display.
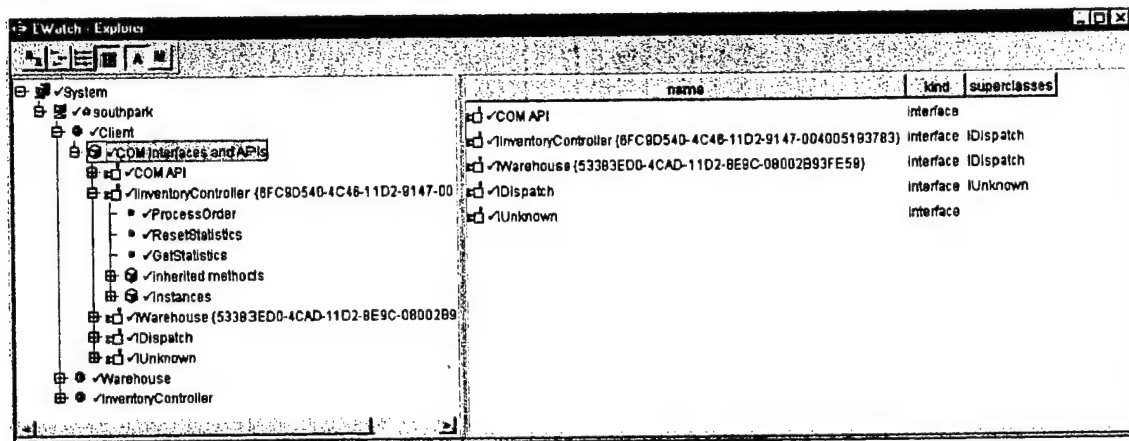
Explorer

Figure 3. EWatch Explorer

The EWatch Explorer (Figure 3) offers a hierarchical view of the elements of the software running on each monitored host. The first job of an EWatch Probe is to report the structure of the software it encounters for the user to view through the Explorer. The view can be telescoped with +/- buttons at each level. The hierarchy for selection of monitoring points and display is:

- System
- Host
- Process
- Interface
- Method
- Instance

For easy identification, the user can associate a symbolic name with each process. The Explorer is the starting point for setting up new monitoring Configurations. Once a Configuration has been found to be useful it can be saved for reuse.
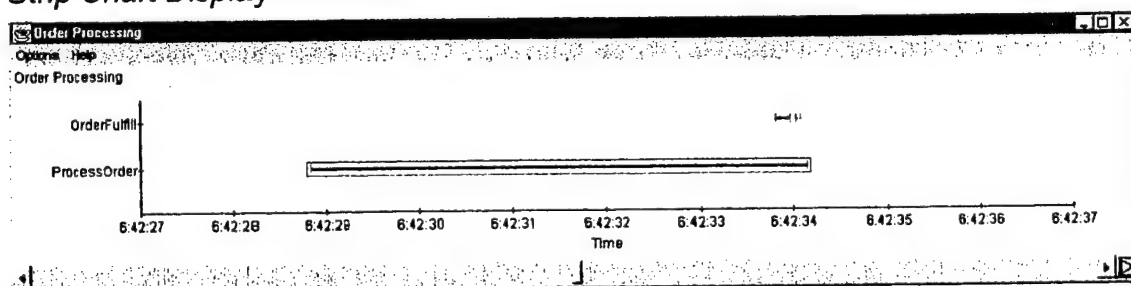
Strip Chart Display

Figure 4. Real-time Strip Chart Display

20

Events are displayed in a time-based, animated Strip Chart that shows application events across time as shown in Figure 4. This intuitive way to observe the behavior of applications can tell the user a great deal about the functioning of the application. Users can zoom in and out on the time scale of the Strip Chart. They can pause and resume the Strip Chart as well as view detail using the Logview. They can scroll back to review a sequence of events and can also change the focus of the review for greater detail or another perspective.

Logview

Figure 5. Information-rich EWatch Logview

Detailed information about events is available in the Logview display (Figure 5). The display can be linked to the Strip Chart so that selecting an event in one will highlight it in the other. Events can be viewed as paired entry-exit events and can be sorted by any column or combination of columns. The Logview columns can show any combination of of:

- Host
- Process
- Thread
- Event Kind
- Event Source

- Method
- Instance
- Parameters & Results
- Time Stamp
- Duration

Sequence Diagram

Users may select a method invocation on a Strip Chart or in a Logview and request a UML-like Sequence Diagram (Figure 6). The Sequence Diagram displays the selected method and all nested method invocations, showing the explicit caller-callee relationships. EWatch can distinguish the appropriate thread in a multi-threaded component and can trace across different component technologies and intervening

21

middleware. Neither the user nor EWatch needs any prior knowledge of the components that comprise a transaction. Once the path of this logical thread is discovered, the user can choose to monitor any other points along the transaction path. If the user invokes
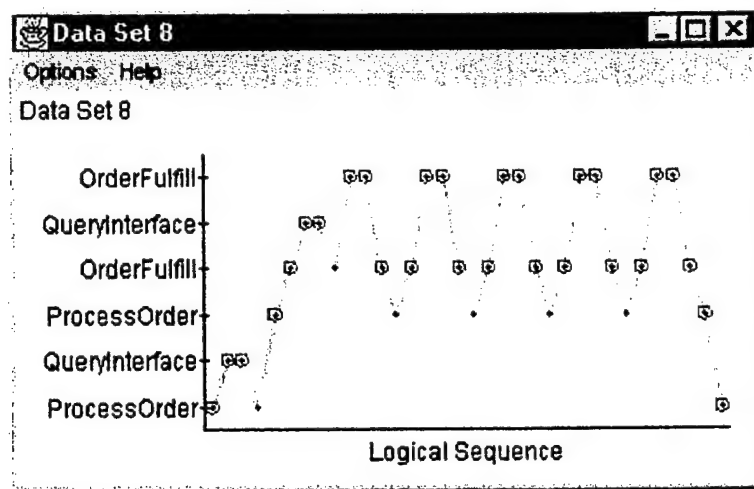
Figure 6. EWatch Sequence Diagram

EWatch's Dynamic Tracing ability, monitoring will take place whenever the selected method invocation initiates a transaction, even when it follows a different path each time the transaction executes.

Portability Through Java

EWatch is written predominantly in Java and can be qualified to run on any machine that supports a Java virtual machine. EWatch Probes are written in a language appropriate to their target. The flexibility of the AEM Framework is shown in the diagram below. New
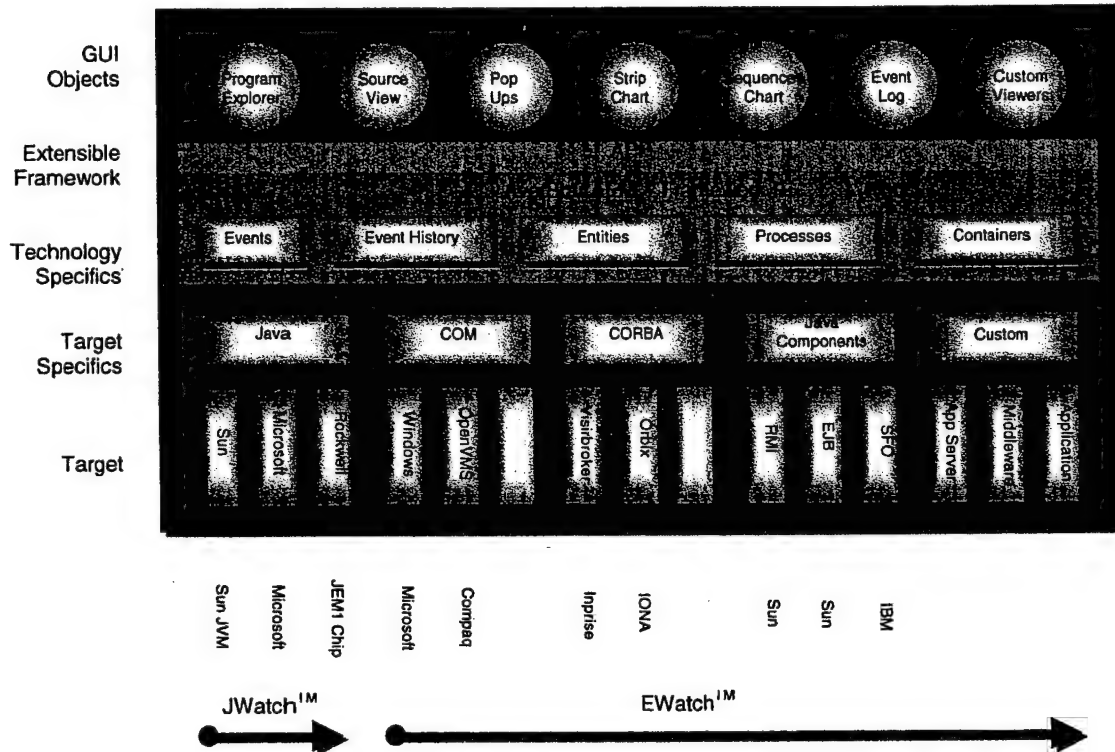
Figure 7. AverStar Application Event Monitoring Framework

GUI objects can be added above the API level. This allows the creation of new viewers and new ways of looking at the data collected by the probes. New probes can be added at the target level. This allows new data to be collected, and new hardware and middleware to be supported.

Approach to Real-Time Monitoring

TRW is currently under Air Force contract to upgrade the Communications, Navigation and Identification (CNI) subsystem of the F-22 Fighter. AverStar has some insight into this program as TRW is using AverStar development tools. The AverStar approach to a real-time monitoring and analysis solution was developed using this program as an example problem.

The CNI subsystem was originally built around TI C30 Digital Signal Processors (DSPs). The Air Force needs higher performance processors to support future requirements, so TRW is currently in the process of re-building the hardware around Analog Devices SHARC chips, and porting the software.

The SHARC processors are much more powerful, and an Ada compiler is available from Analog devices, so porting the existing code (more than 200,000 lines) for the CNI application is reasonably tractable. However, every aspect of the application timing will change in the process.

Substituting the more powerful DSP does not make everything uniformly faster; it speeds up some modules more than others. Furthermore, the extreme pipelining and parallelism found in DSPs makes predicting the timing changes very difficult. In a simple, mono-tasking application, this would not matter, but the F-22 CNI software is anything but simple. It contains many Ada tasks executing asynchronously of one another; it interfaces with several Application-Specific Integrated Circuits (ASICs) to communicate with the rest of the avionics suite; and it must communicate with various tasks and processors in the flight control and other F-22 subsystems as well.

Minimally intrusive monitoring for Real-Time Applications

Our primary technique involves inserting probes in all of the middleware, including COM, ORB(s), and other Internet infrastructure. Each probe transmits a small packet of data to a monitoring machine on the network each time it fires or is activated. The monitoring machine produces multiple tabular and graphic displays.

F-22, like most real-time embedded applications, doesn't use CORBA, Java Beans, DCOM, or Windows '95. It uses a "hard real-time" embedded operating system and program-specific interfaces to ASICs and other I/O devices which are custom built by TRW. Applying the EWatch architecture to CNI would require instrumenting its middleware: the real-time executive, runtime systems, and I/O drivers. With probes in place in its middleware, the CNI application would generate data that can then be displayed and analyzed by the same EWatch tool now in use for desktop systems.

Most real-time applications cannot simply send packets of results from the probes across a general Internet protocol. The timing analysis and verification must be done in the real flight configuration to be valid, and this configuration does not have spare computers, spare bandwidth, or a general message logging facility. To properly measure F-22 CNI

timing, we would have to modify the probes to save data in local memory and create a separate program to collect and collate the data for post-processing. This is exactly the approach we used successfully on the MWave Development Tools for IBM.

The updated CNI hardware has 512k of memory on each SHARC chip, and an additional 1.5 megabytes of off-chip memory for each SHARC. Fully instrumenting the code will result in the order of 100,000 probes per second, which would fill up all available memory in only a few seconds. This is typical of embedded applications in general. As a result, the baseline for F-22 is to instrument only the low level, and to do the testing in several short runs with different objectives, instead of leaving the instrumentation active on all of the low-level routines all the time.

We have designed a dynamic compression technique to solve this problem, making it possible to record many probe events in a small memory buffer. The figure below illustrates the intended use.

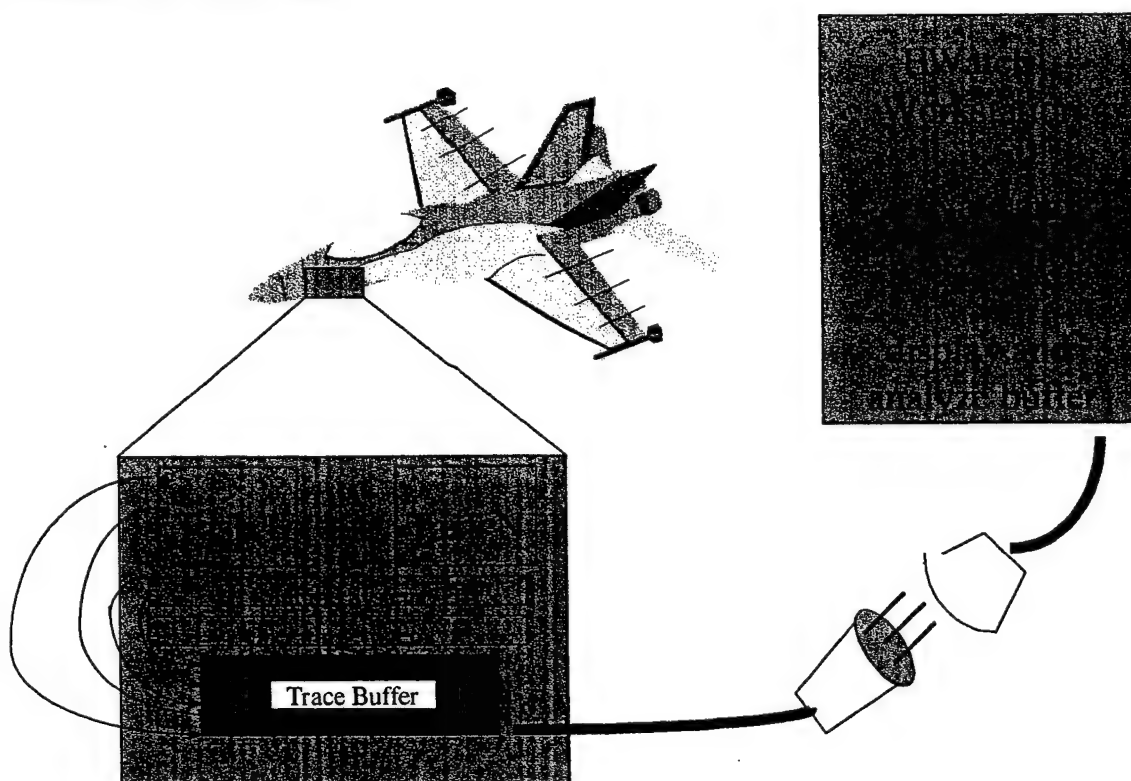**Figure 2: Real-time Traces for Embedded Applications**

We would connect a debugging workstation to the real-time system to activate and deactivate probes. Then we will disconnect the workstation and embed the CNI system in the full avionics suite or the aircraft itself and run the application. After the run, we will reconnect the workstation and extract the buffer of probe data, for off-line analysis and display.

Dynamic Compression of Traces

As part of our EDCS investigations, we have created a new approach to managing and analyzing probe data that eliminates the need for either a large storage buffer or direct connection to the monitoring workstation. That approach would enable us to instrument the whole application and do much more sophisticated analysis.

Probes generate little packets of information, giving the id of the routine being called, and sometimes additional items such as the id of the currently active task or a parameter passed to the routine. The raw data looks like a sequence of procedure ids – A, B, C, B, C, D, A, B, C, B, C, D – that correspond to the sequence in which routines are called. Another way of representing the sequence above is $(A, (B, C)^2, D)^2$. The latter notation is much more compact. In fact, the larger and more complex the sequence, the more savings are gained by the compact notation.

Real applications consist of loops within loops within loops. These are not generally explicit loops in the source code – they are logical loops through the procedures. A real application will generate at least tens of thousands of procedure calls per second, but the whole execution can often be reduced to the repetition of one loop by each task. If an application is truly synchronous and cyclic, the compact representation of an infinitely long execution profile will be a small, finite sequence that repeats indefinitely.

We can convert the raw sequence of procedure calls to the compact form in real time as the data is created. Furthermore, our algorithm for finding the repeated sub-sequences uses very little additional data during the computation, so we could use this technique to squeeze the whole execution trace of F-22 CNI code into the small available memory.

If further compression seems warranted, we could go further along this route by providing a form of "lossy compression". Three example techniques we considered are:

Putting ranges on loops. If an inner loop executes a different number of times on each iteration of the outer loop, we can still collapse both loops by storing a range of iterations on the inner, yielding a notation like $(A \ B \ (C \ D)^{5\text{-}10} \ E)^6$, meaning that the outer loop executed six times, with the inner loop executing five to ten times on each iteration of the outer loop. We lose the details (which iteration of the outer loop caused the most iterations of the inner, etc.), but we still gain an interesting and useful model of the application.

Identifying "procedure calls". In the sequence A B C D F G C D H, there is no loop, but we could treat the repeated sequence C D as a subroutine, yielding a notation like:

A B [C D] F G* H
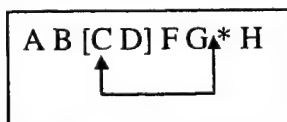
Identifying conditionals. This conversion is the most difficult, but much stronger compression will result if we can detect loops that are almost the same, yielding notations like $(A \ B \ \{C \ \textbf{or} \ D \ E\} \ F)^3$.

Analysis and Display

The compact form of an execution trace is in itself a model of the application. We can examine it to better understand the dynamics of the program. Furthermore, we can:

- Compare models after a change to determine if, and where, the execution profile has changed
- Compare a model with the corresponding design documents to see if they match
- Pre-compute stack requirements based on actual call profiles instead of worst case profiles
- Identify potential race conditions and deadlocks before they occur
- "Roll up" execution times for low level functions to the higher level application components that are ultimately responsible for them being invoked
- Estimate nominal and worst-case processor utilization and check for deadline feasibility via Rate Monotonic Analysis.
- Compute the margin available for further growth in each component

While this approach has not been put into practice on the F22 or any other program, we feel that it is a unique approach that could have a large impact on mission-critical distributed real-time programs.

AppletMagic Programming Environment
Overview

The AppletMagic Programming Environment provides an integrated set of tools for creating and evolving dynamic, distributed, heterogeneous systems based on Ada 95, Java, and CORBA and COM components.

The Ada/Java Execution Environment currently consists of the AdaMagic™ Ada 95 compiler front end, a Java bytecode (J-code) code generator, and a Java interpreter and development environment. It is the first non-Java environment to take advantage of the dynamic behavior and portability that Java has made popular in many different application domains. There is complete interoperability between Ada-generated and Java-generated J-code, the combination of the two being particularly well suited to the evolutionary design and implementation of complex systems.

The Distributed Application Monitor, code-named JADE, allows analysis, debugging, and tuning of distributed, heterogeneous, component-based systems. JADE shows the J-code applets and applications that are executing, where they are, the threads they are running, and system load information. JADE also allows conventional debugging of applets and applications, whether or not source code is available for them. JADE is being enhanced to show the message traffic between software components, including those built using the CORBA and COM component object models.

The Ada/Java Execution Environment

AppletMagic is based on a complete mapping between Ada 95 and Java, and supports all but those aspects requiring significant out-of-line code (such as record assignment and protected types). The process of defining this mapping has reinforced our early impression that Java and Ada 95 are semantically very compatible. The combination of the two is remarkably seamless, and loses essentially nothing while gaining the best aspects of both languages. Ada 95 provides significantly more compile-time checking, as well as a number of nice compile-time features such as enumeration types, generic templates, and user-defined operators, while Java provides automatic garbage-collected storage-management, platform independence of both J-code and its rich class library, and integration with the World Wide Web.

We have found the combination of the two technologies to be uniquely productive. Writing in Ada 95 provides the same functionality as Java, with enhanced error checking and reliability. For example, for one applet that we translated, after implementing full constraint checking in the Ada to Java compiler, we immediately began to reap the benefits of Ada's notion of range checks. The first Constraint_Error that was raised identified a small error in the hand translation from the original Java applet into Ada.

After fixing that error, the next Constraint_Error identified a deep logic error in the original Java applet, which ultimately turned out to be due to the randomization mechanism used to initialize the application.

Our early experience has confirmed our belief that the combination of Ada 95 and Java is an excellent technology for the evolutionary design and implementation of complex systems, benefiting from the rapidly growing commercial support for Java, and the excellent software engineering advantages of Ada.

The Distributed Application Monitor (JADE)

JADE is a multi-faceted tool. It is a monitor for distributed component applications, as well as being the first true multi-process source-level debugger for Java (already the language of choice for many component developers and integrators).

JADE treats the components of a distributed application as a set of interacting objects each sending and receiving messages. This is consistent with the fact that components in many distributed component applications are black boxes - there is no source available,
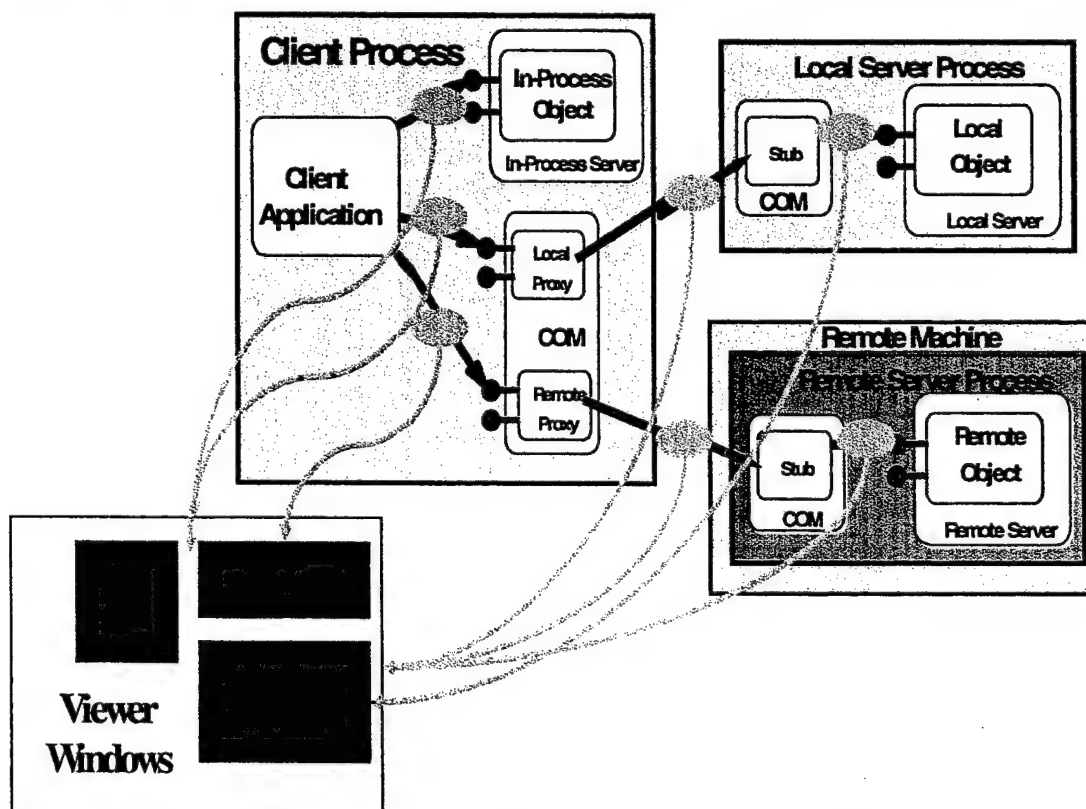
so the only description we have of their behaviors is the message traffic between them. By placing probes that "listen" to messages at key points in the distributed application,

JADE can interpret and display the messages in a time-correlated viewer so that developers and support staff can gain an understanding of the state of the application when it fails to work as expected. And JADE's extensible, event-driven framework allows for custom-made, application-specific probes and viewers to be plugged in to provide a uniquely-tailored view of the behavior of distributed, heterogeneous, component-based systems.

As a Java debugger, JADE provides two different views of the application under test: the Source Viewer and the Program Explorer. The Source Viewer, of which there can be multiple instances, displays the source code of the selected process; it also allows the user to set and clear breakpoints and to single step the currently active thread, stepping into or over calls. The Program Explorer, modeled after Microsoft's Windows Explorer, displays the processes under test, from a static and dynamic perspective. From the static perspective, a process consists of packages, classes, methods, and class static fields; from the dynamic perspective, a process consists of thread groups, threads, stack frames, frame-resident variables and parameters, and class instance fields.
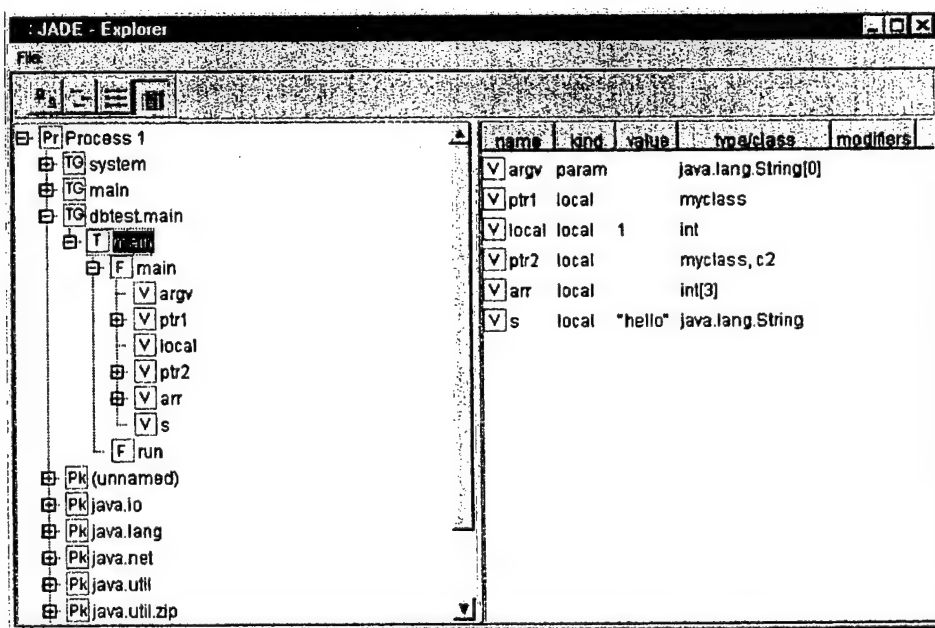
The Program Explorer also enables the user to debug effectively, even in the absence of source code. The static and dynamic views continue to show the same source-level information, and the user is able to control program execution by setting breakpoints on method invocations.

Jade Requirements Specification

(version 2, 10/23/96)

[ed. note: This document is the original requirements specification. The software was written in Java using the AWT]

Introduction

The main technical goals of the Applet Distributed Monitor and Debugger (code named "Jade") are:

- Provide capabilities for debugging and monitoring;
- Allow debugging of distributed programs;
- Work with programs written in AdaJava (applets and applications) and also programs written in Java;
- Be highly adaptable and extensible

Important design decisions

GUI: The two main choices are to make the graphical user interface that resembles a Windows program, or one that resembles a Web browser. As much as possible we will choose to make the debugger look like a native Windows program. The Symantec Café debugger looks like a native Windows application, and the Sun Java Workshop debugger looks like a browser. In our opinion the Symantec debugger looks better.

Technology used to build the GUI: Here the main choices are Microsoft Foundation Classes (MFC), XVT or a similar program (XVT is a system independent GUI library that has been used to build the IMSI Passkey debugger), or Java.

MFC would be the best way to make a native-looking Windows program, and there are now portability libraries that let MFC programs be ported to the Sun and other systems.

The debugger when targeting a Java Virtual Machine (JVM) will be implemented in top of a Java package sun.tools.debug (see next section on debugger architecture) and this argues for using Java as the development environment. It is possible to interface between C++ and Java, but at this time it is extremely difficult, and it would increase the amount risk in implementation.

XVT seems to have worked well for IMSI in the past, but again, using XVT implies a C-based system, and then having to interface between C and Java.

Currently the best approach seems to be to implement the GUI in Java. A reasonable GUI can be produced, although it will not look as Windows-native as it would if implemented in MFC. Doing the project with Java means using the Java Abstract Windows Toolkit (AWT). This is good for portability, but limiting in other ways. For example, making a source window with lines highlighted in a different color or font, or with icons like stop signs for breakpoints embedded in the text, is difficult or impossible
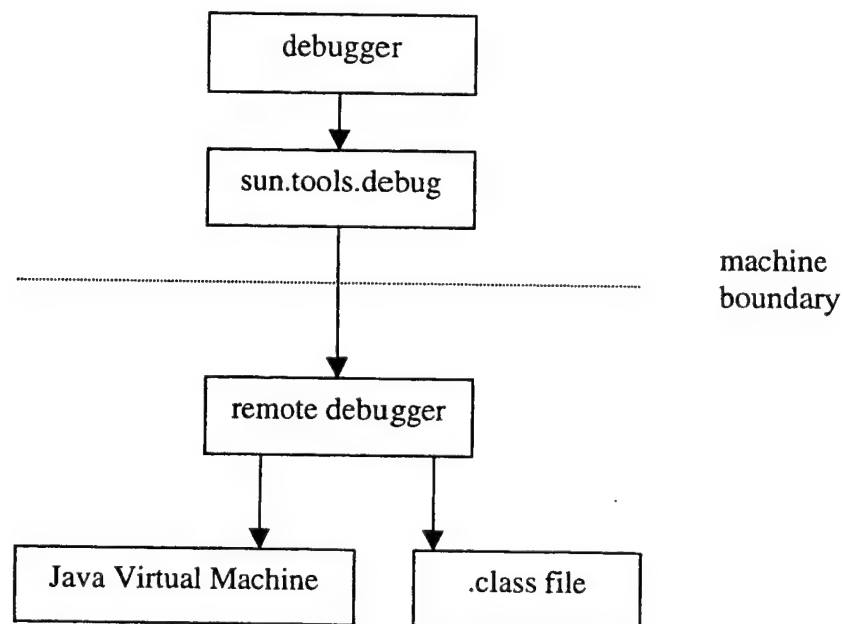
with the current AWT. However, the AWT is still pretty new and is sure to be enhanced in future versions.

Languages and environments supported: The first version of the debugger must support debugging programs written in Java and AdaJava, and targeting the JVM, for applets and applications. It must work for distributed programs, that is, applets and applications running in separate virtual machines, and potentially on separate computers.

Windows is assumed to be the primary host system for the debugger, although writing in Java gives a lot of portability in this area.

Java debugger architectures

Java debuggers have this architecture:

```
                    ┌─────────────────────┐
                    │      debugger       │
                    └─────────────────────┘
                              │
                              ▼
                    ┌─────────────────────┐
                    │   sun.tools.debug   │
                    └─────────────────────┘
                              │
─────────────────────────────│─────────────────────        machine
                              │                              boundary
                              ▼
                    ┌─────────────────────┐
                    │   remote debugger   │
                    └─────────────────────┘
                       │               │
                       ▼               ▼
          ┌─────────────────────┐  ┌─────────────────────┐
          │ Java Virtual Machine│  │    .class file      │
          └─────────────────────┘  └─────────────────────┘
```

A debugger like Sun JDB (a text-mode debugger that comes with the JDK, including source code), or the Sun Java Workshop debugger, or the Symantec Visual Café debugger, does its work through the package sun.tools.debug, which actually performs most of the debugging actions. This package is capable of managing breakpoints, managing threads and objects and classes on the target JVM, and reading and setting data values. It creates a remote debugger to do this work. If the target that is being debugged is on a different machine than the host where the debugger is running the remote debgger executes on the target and the communications are managed internally inside sun.tools.debug. The debugger doesn't have to read a program's symbol table, all the information is in the .class file, as seen through the sun.tools.debug interfaces.

From the point of view of a debugger writer, the sun.tools.debug interface is very high level and makes writing a debugger easy. The JDB sources can be printed out on just 14 pages. This package includes most of the functionality a debugger would want, but three significant things seem to be missing:

- Data watchpoints;
- Calling a method (subprogram) in the target, or changing the target's program counter;
- Low level debugging (addresses, bytes, registers etc.)

Our approach will be:

- Make a package of operations that a debugger can call, including functionality in sun.tools.debug;
- Add other functionality like address operations, registers, and data watchpoints;
- For all functionality areas, include a boolean function that says whether the functionality is implemented on the current target.

Debugger requirements

Basic debugging: the debugger will support at least all the functionality now in JDB (see section *JDB Commands*).

Distribution – basic support: the user will be able to start one or more programs, locally or on a remote machine, and then run the programs, setting breakpoints, and looking at threads, data, etc. for each program. If something goes wrong with a distributed program running under the debugger, the user will be able to see the state of each piece.

Distribution – advanced support: the debugger would understand a communications method between distributed Java programs, for example, Java Beans (if that method becomes popular). The debugger would have some built-in understanding of the distribution model and the calls that are made for sharing objects and communications and synchronization between programs, and the states the programs can be in. For example, if Java Beans has a high level operation which requests access to an object, and this high level operation is implemented with a low level socket call, and the program is blocked waiting for the object/socket, the debugger would understand something about this.

This will be achieved through the debugger's extensibility. The basic debugger will not understand about a communication method like Java Beans, but an extension can be written that does.

Ada support: there is a name mangling that goes on when translating an Ada source program to a set of Java classes and objects that execute on the target JVM. The debugger will have to understand this mainly for data references. The debugger core will be able to find and display and modify values of data, based on the data's name in the target environment. A debugger extension will understand translating user-supplied names in a source language to and from names on the target machine. This plug-in will

be close to a null translation for Java programs, but it will have a lot of work to do for Ada programs.

A little bit of support is also needed for Ada inlining, which is not a Java features. The Java model is that each .class file comes from one source file, but this doesn't work for inline subprograms and inline generics.

Monitoring: sun.tools.debug doesn't give too much monitoring functionality. The debugger is notified when a thread dies, or the program exits, or a breakpoint, or an exception is executed. It can look at the current data and objects and threads. It can ask how much memory the program is using.

One thing we could do is have a thread inside the debugger that at a regular interval collects information on the state of the remote program(s), and reports its information graphically.

Extensibility: the debugger should be easily extensible for other languages and target machine architectures. For example, when working with a memory mapped device, a customer should be able to write a small debugger extension which brings up a window and shows the device's bits and bytes graphically.

If the extensibility is well done, lots of different things could be added easily, and the debugger could become popular because third parties find useful ways to extend it (like open source). To realize this idea, the debugger" services should be a publicly visible API, e.g., a Java package with public classes and methods, which a new debugger plug-in can call and be called-back by, and extend or override as necessary. Also each service should include a boolean function saying whether it is available. So for example, if the debugger has a "registers" or "memory" window and the debugger is targeted to a JVM where these services are unavailable, these windows can gray themselves out in the menu or otherwise fail gracefully, while other services continue to work.

Core GUI and core services

The core GUI contains:

- Main window, main menu and buttons with some built-in features. Debugger extensions can install more options into the main menu and can add buttons to the main window. When the main window is minimized, restored, or killed, other debugger windows are minimized, restored or filled respectively.

- The GUI has a source view, a data view, and a view of programs/threads/stacks as built-in window types. Each one of these comes up as a separate window. Each of these views has a default behavior that can be overridden, and has built-in menus and buttons that can be added to.

- The debugger will have a way to distinguish between multiple programs.

- Other views can be built as debugger extensions: assembly view, memory view, register view, monitor view, code coverage view etc.

The debugger has a set of core services:

- It implements a package of debugging interfaces for program control and symbol table that is a superset of sun.tools.debug. For each functionality area, there is a function saying whether that area is supported.
- Each action where a service is requested from a debugging interface has a textual command. Users can optionally see the commands that are being generated and can capture these commands into scripts, and can give those scripts names with a macro-like capability.
- The debugger maintains lists of callbacks, so windows and extensions can be notified of events in which they have a registered interest.
- The debugger maintains a notion of context, for each program being debugged, including current thread in program, and source file and line and stack in thread.
- For each stack frame the debugger knows the source file and language.
- New menu items and buttons can be added to the main window and to other windows. When these items are selected by the user the attached code is called by the debugger.

JDB Commands

Here is the current JDB command summary:

```
Threads [threadgroup]
thread <thread id>
suspend [thread id(s)]
resume [thread id(s)]
where [thread id] | all
threadgroups
threadgroup name

print <id> [id(s)]
dump <id> [id(s)]

locals

classes
methods <class id>

stop in <class id>.<method>
stop at <class id>:<line>
up [n frames]
down [n frames]
clear <class id>:<line>
step
cont
```

```
catch <class id>
ignore <class id>

list [line # | method]
use [source file path]

memory
gc

load <class name>
run <class> [args]

!!
help (or ?)
exit (or quit)
```

MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of Information Systems Science and Technology to meet Air Force unique requirements for Information Dominance and its transition to aerospace systems to meet Air Force needs.